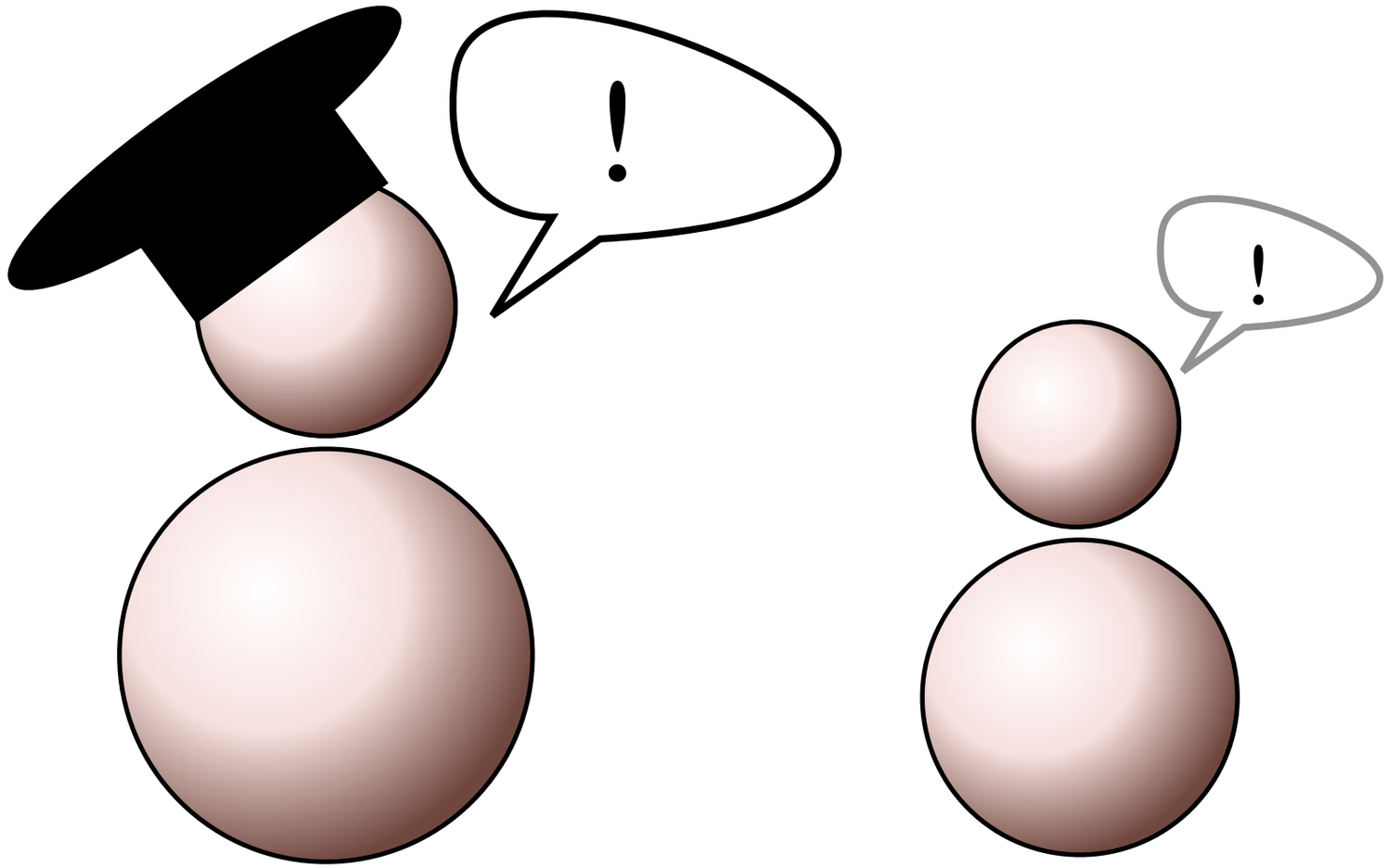
Reinforcement Learning

Image: Wikipedia

“Supervised learning”
(most neural network applications)

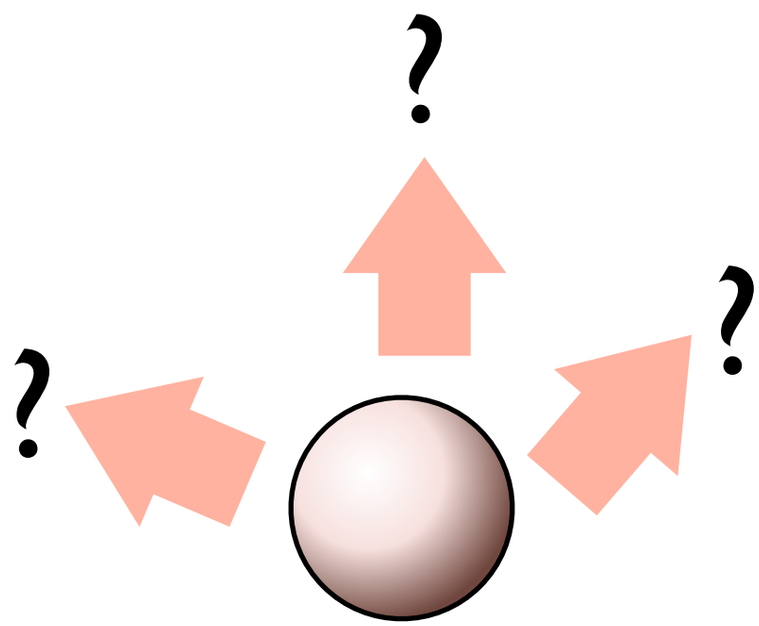


teacher
(smart)

student
(imitates teacher)

final level limited by teacher

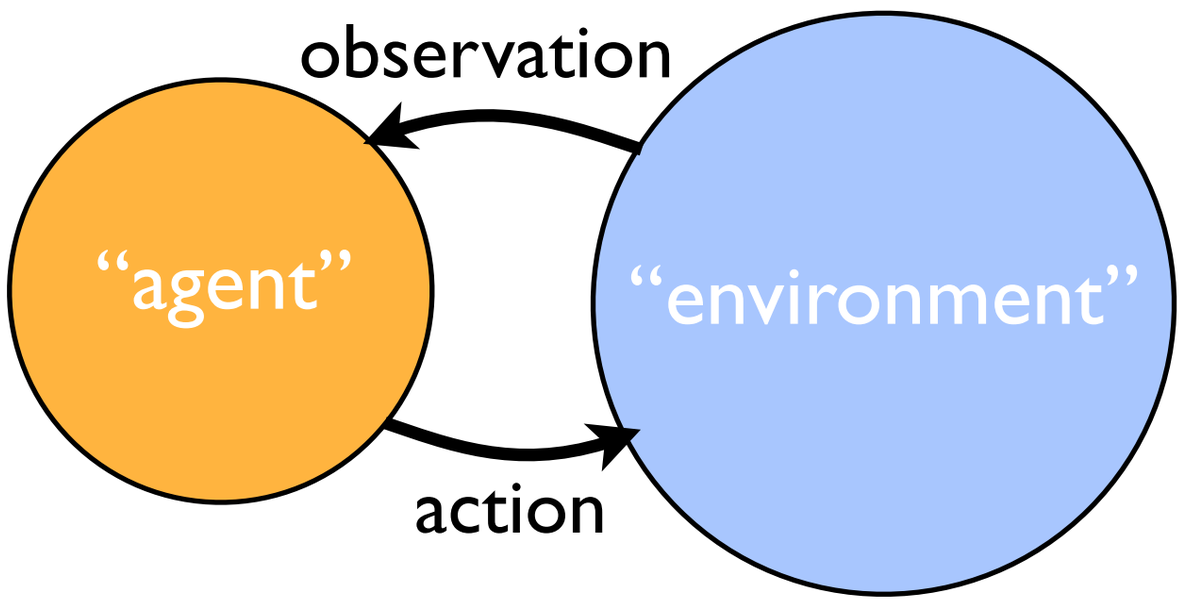
“Reinforcement learning”



student/scientist
(tries out things)

final level: unlimited (?)

Reinforcement learning



fully observed vs. partially observed "state" of the environment

Self-driving cars, robotics:
Observe immediate environment & move
Games:
Observe board & place stone
Observe video screen & move player

Challenge: the "correct" action is not known!
Therefore: no supervised learning!

Reward will be rare (or decided only at end)

Reinforcement learning

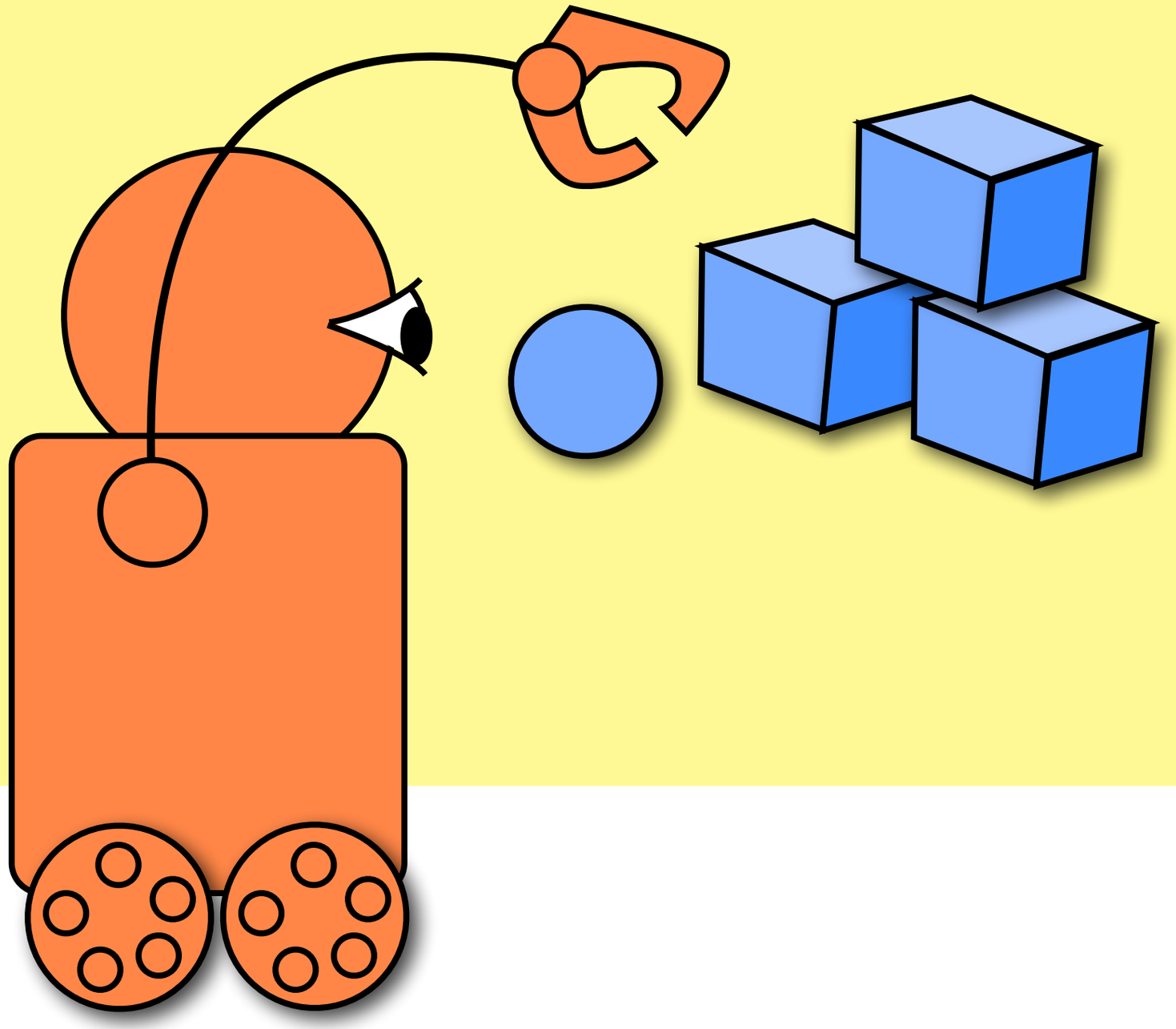
Use **reinforcement learning**:

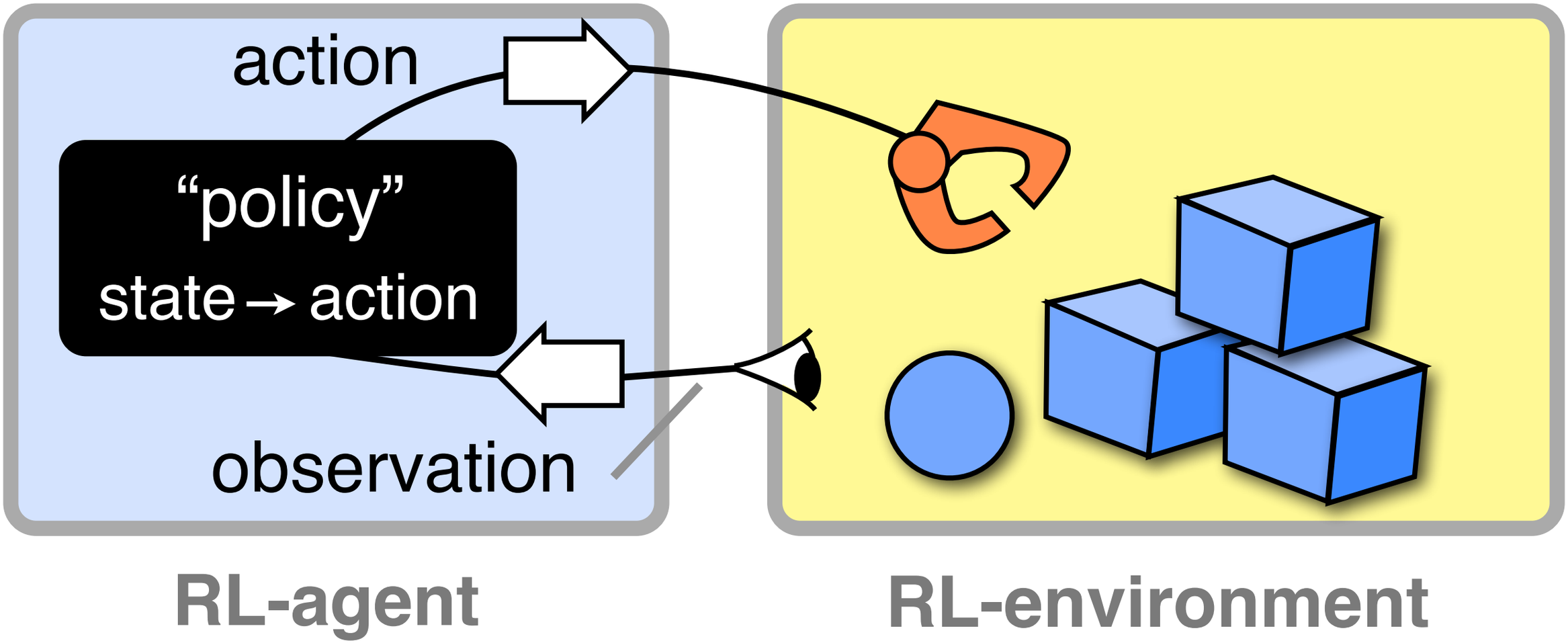
Training a network to produce actions based on rare rewards (instead of being told the 'correct' action!)

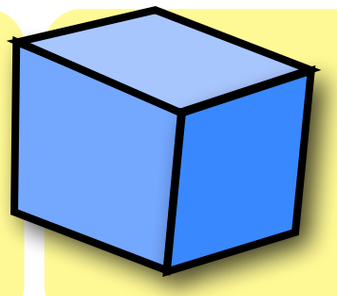
Challenge: We could use the final reward to define a cost function, but we cannot know how the environment reacts to a proposed change of the actions that were taken!

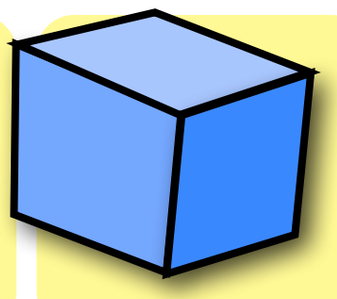
(unless we have a model of the environment)

Reinforcement Learning: Basic setting

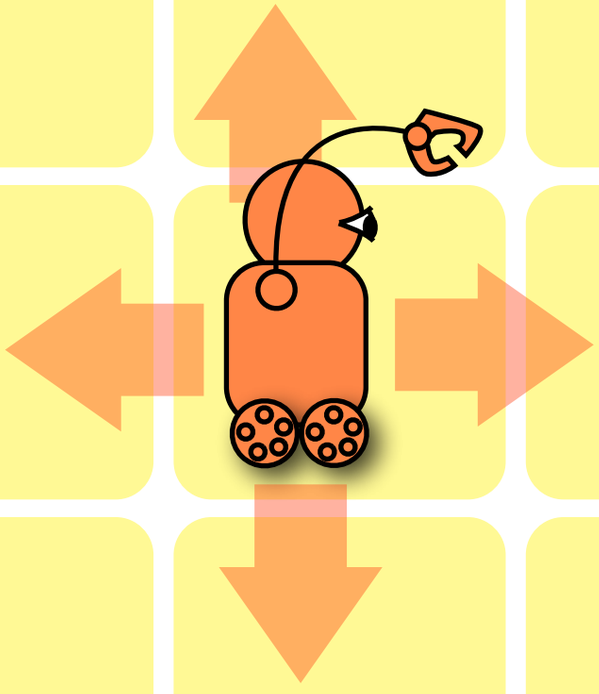


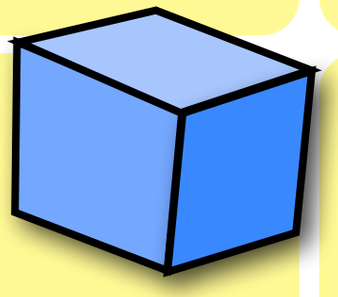

action
“policy”
state → action
observation
RL-agent
RL-environment

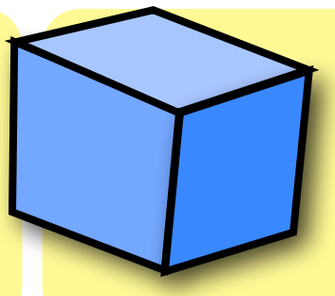


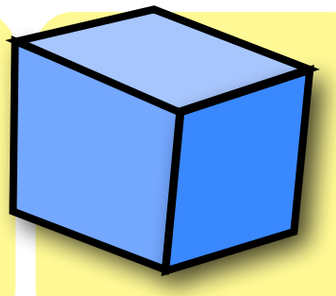


state = position x,y
action = move (direction)

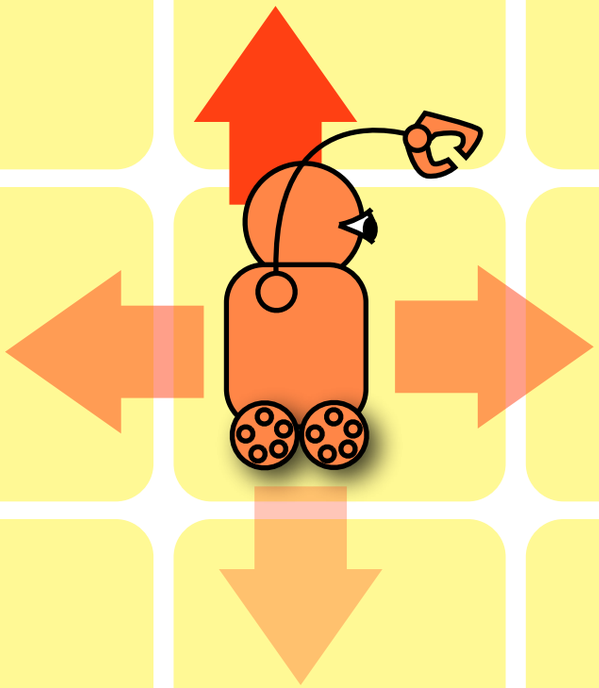


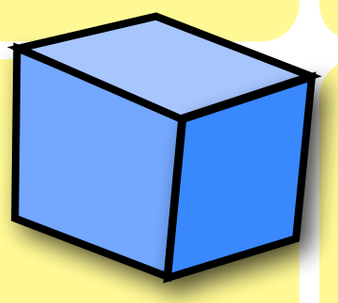






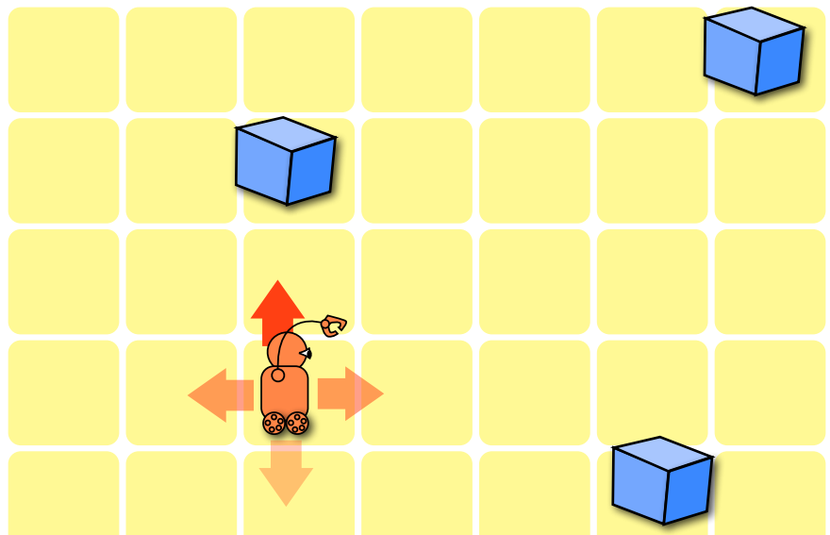
state = position x,y
action = move (direction)
reward for picking up box





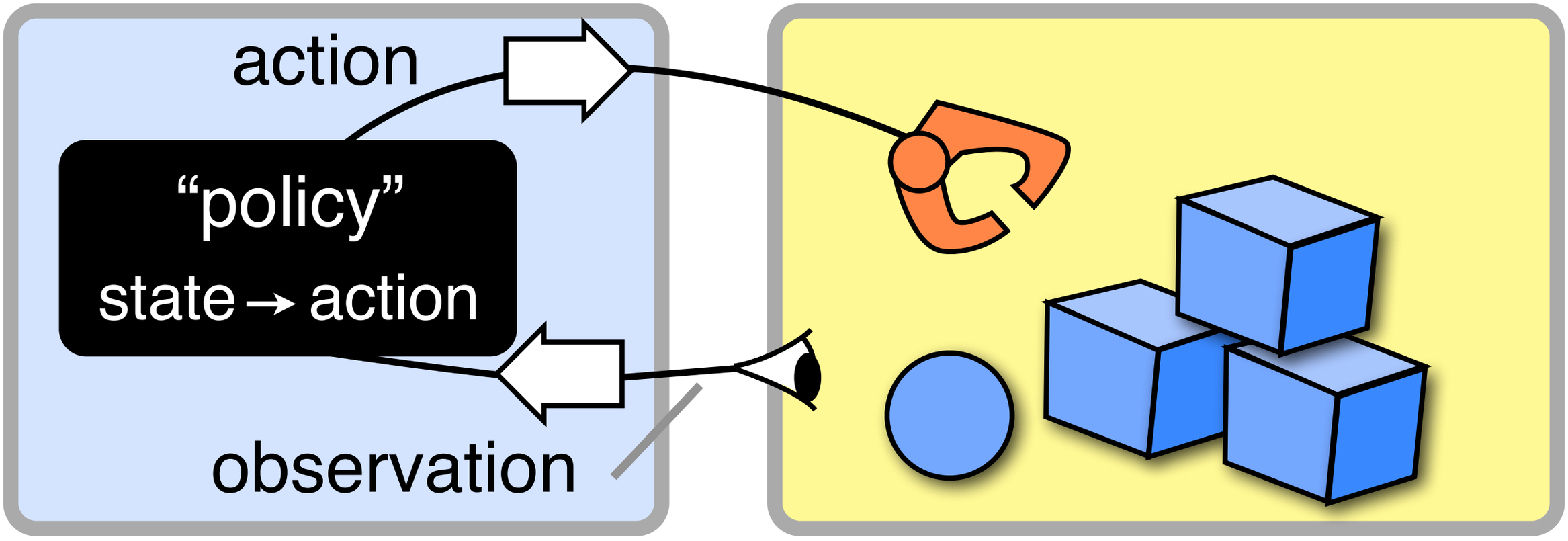
Policy Gradient

=REINFORCE (Williams 1992): The simplest **model-free** general reinforcement learning technique

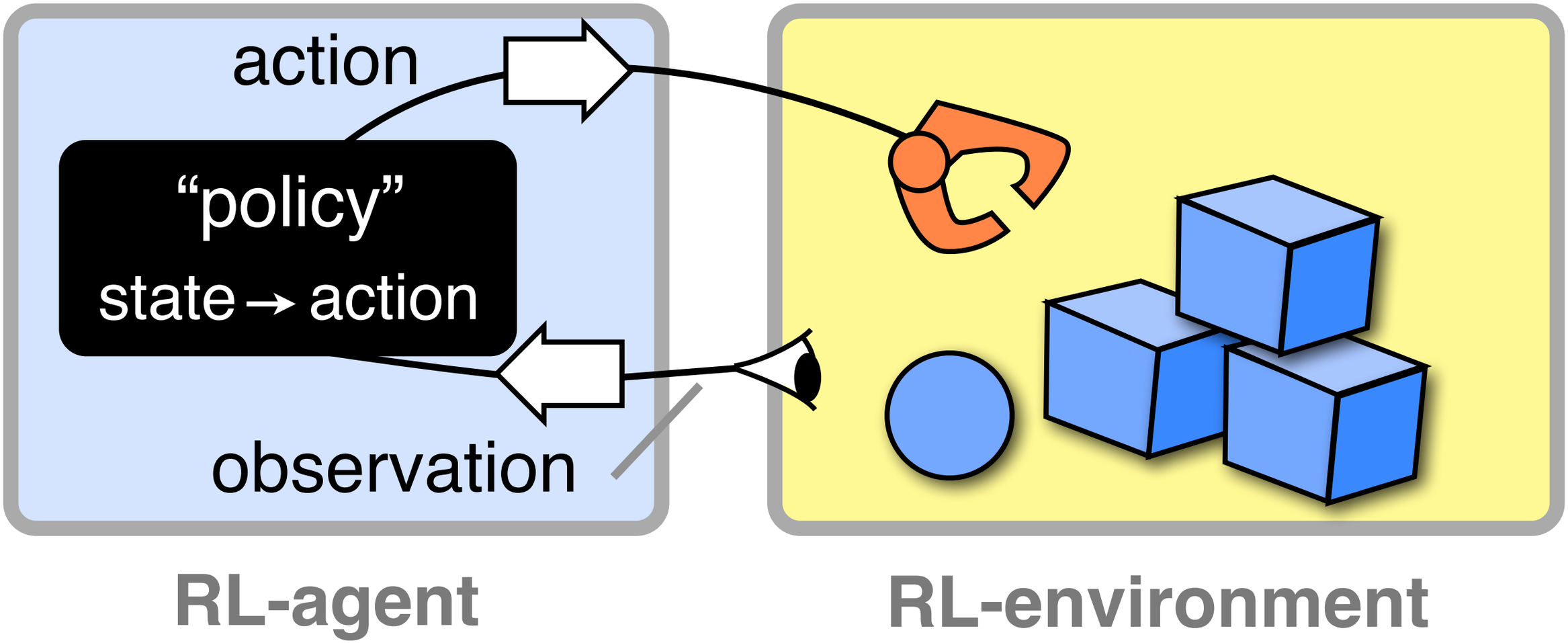


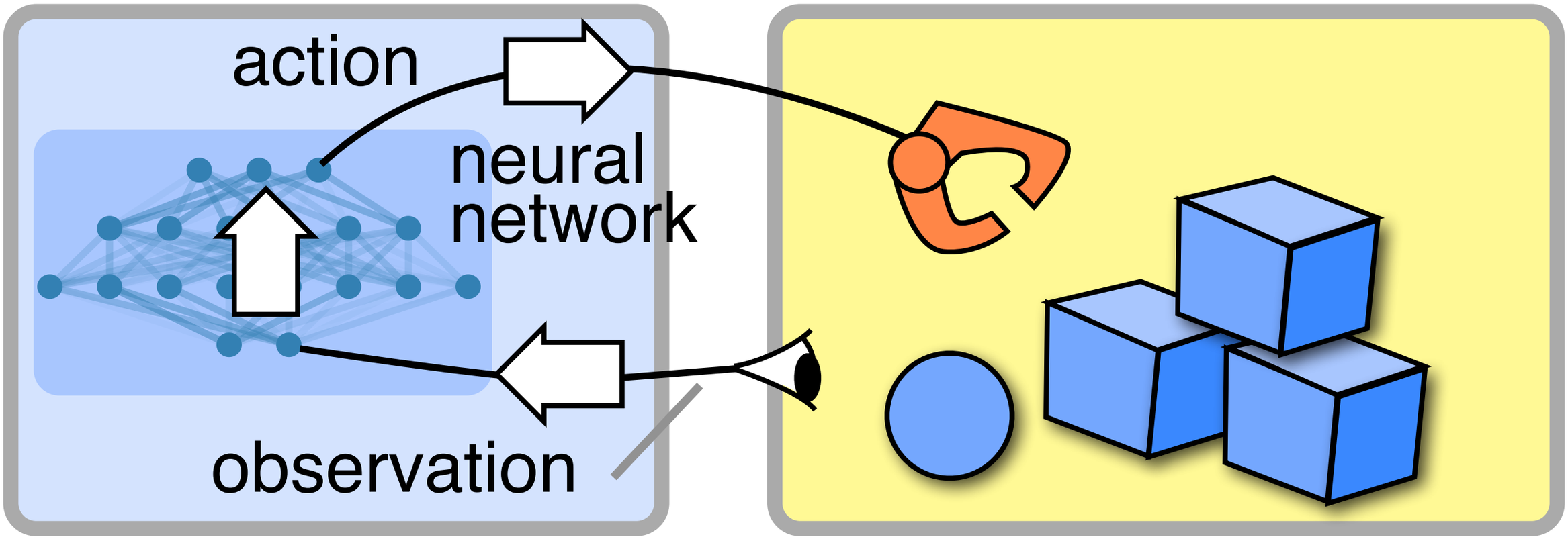
Basic idea: Use probabilistic action choice. If the reward at the end turns out to be high, make **all** the actions in this sequence **more likely** (otherwise do the opposite)

This will also sometimes reinforce 'bad' actions, but since they occur more likely in trajectories with low reward, the net effect will still be to suppress them!



Policy: $\pi_\theta(a_t|s_t)$ – probability to pick action a_t given observed state s_t at time t


action
“policy”
state → action
observation
RL-agent
RL-environment


action
neural
network
observation
RL-agent
RL-environment

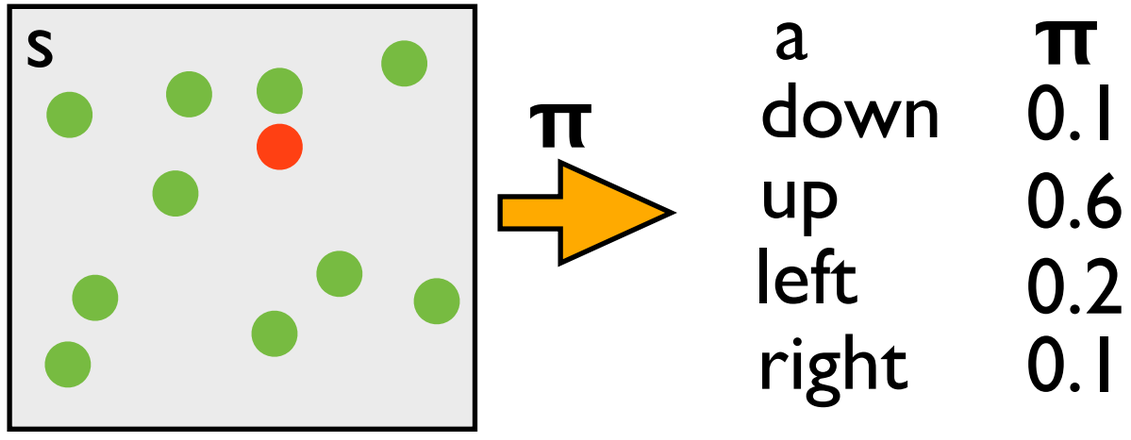
Policy Gradient

Probabilistic policy:

Probability to take action a, given the current state s

$$\pi_\theta(a|s)$$

parameters of the network



Environment: makes (possibly stochastic) transition to a new state s', and possibly gives a reward r

Transition function $P(s'|s,a)$

Policy Gradient

Probability for having a certain trajectory of actions and states:

product over time steps

$$P_\theta(\tau) = \Pi_t P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$$

trajectory: $\tau = (\mathbf{a}, \mathbf{s})$
$\mathbf{a} = a_0, a_1, a_2, \ldots$
$\mathbf{s} = s_1, s_2, \ldots$ (state 0 is fixed)

Expected overall reward (='return'): sum over all trajectories

$$\bar{R} = E[R] = \sum_\tau P_\theta(\tau)R(\tau)$$

— return for this sequence (sum over individual rewards r for all times)

sum over all actions at all times and over all states at all times >0

$$\sum_\tau \cdots = \sum_{a_0,a_1,a_2,\ldots,s_1,s_2,\ldots} \cdots$$

Try to maximize expected return by changing parameters of policy:

$$\frac{\partial \bar{R}}{\partial \theta} = ?$$

Policy Gradient

$$\frac{\partial \bar{R}}{\partial \theta} = \sum_t \sum_\tau R(\tau) \underbrace{\frac{\partial \pi_\theta(a_t|s_t)}{\partial \theta} \frac{1}{\pi_\theta(a_t|s_t)}}_{\frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}} \Pi_{t'} P(s_{t'+1}|s_{t'}, a_{t'}) \pi_\theta(a_{t'}|s_{t'})$$

Main formula of policy gradient method:

$$\frac{\partial \bar{R}}{\partial \theta} = \sum_t E[R \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}]$$

Stochastic gradient descent:

$$\Delta\theta = \eta \frac{\partial \bar{R}}{\partial \theta}$$

where $E[\ldots]$ is approximated via the value for one trajectory (or a batch)

Policy Gradient

$$\frac{\partial \bar{R}}{\partial \theta} = \sum_t E[R \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}]$$

Increase the probability of all action choices in the given sequence, depending on size of return R.
Even if R>0 always, due to normalization of probabilities this will tend to suppress the action choices in sequences with lower-than-average returns.

Abbreviation:

$$G_k = \frac{\partial \ln P_\theta(\tau)}{\partial \theta_k} = \sum_t \frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta_k}$$

$$\frac{\partial \bar{R}}{\partial \theta_k} = E[RG_k]$$

Policy Gradient: reward baseline

Challenge: fluctuations of estimate for return gradient can be huge. Things improve if one subtracts a constant baseline from the return.

$$\frac{\partial \bar{R}}{\partial \theta} = \sum_t E[(R - b)\frac{\partial \ln \pi_\theta(a_t|s_t)}{\partial \theta}]$$
$$= E[(R-b)G]$$

This is the same as before. Proof:

$$E[G_k] = \sum_\tau P_\theta(\tau)\frac{\partial \ln P_\theta(\tau)}{\partial \theta_k} = \frac{\partial}{\partial \theta_k}\sum_\tau P_\theta(\tau) = 0$$

However, the variance of the fluctuating random variable (R-b)G is different, and can be smaller (depending on the value of b)!

Optimal baseline

$$X_k = (R - b_k)G_k$$

$$\mathrm{Var}[X_k] = E[X_k^2] - E[X_k]^2 = \min$$

$$\frac{\partial \mathrm{Var}[X_k]}{\partial b_k} = 0$$

$$b_k = \frac{E[G_k^2 R]}{E[G_k^2]}$$

$$G_k = \frac{\partial \ln P_\theta(\tau)}{\partial \theta_k}$$

$$\Delta\theta_k = -\eta E[G_k(R - b_k)]$$

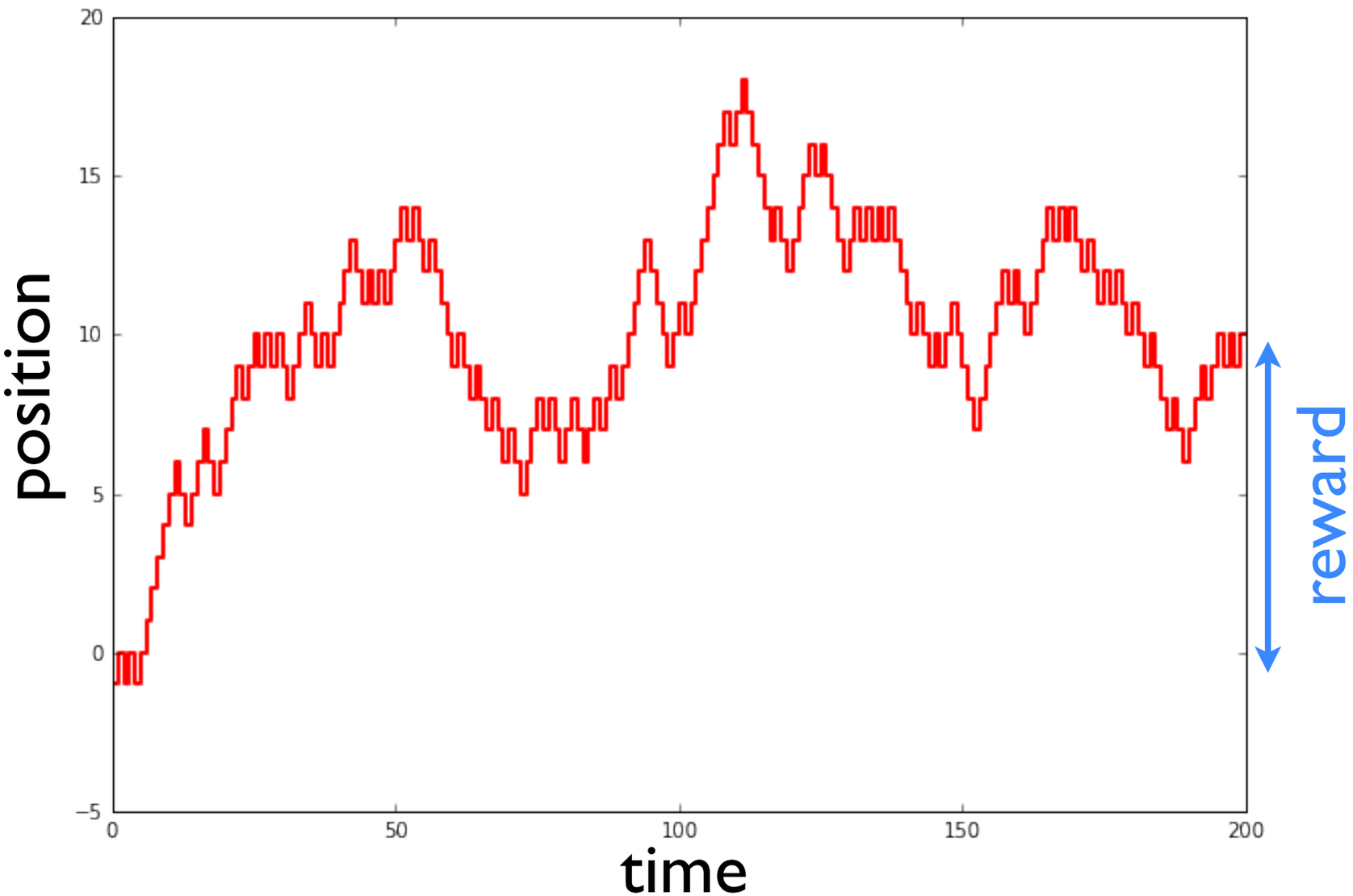
For more in-depth treatment, see David Silver's course on reinforcement learning (University College London):

http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

The simplest RL example ever

A random walk, where the probability to go "up" is determined by the policy, and where the return is given by the final position (ideal strategy: always go up!)

(Note: this policy does not even depend on the current state)



The simplest RL example ever

A random walk, where the probability to go "up" is determined by the policy, and where the return is given by the final position (ideal strategy: always go up!)

(Note: this policy does not even depend on the current state)

policy $\pi_\theta(\mathrm{up}) = \dfrac{1}{1+e^{-\theta}}$ **return** $R = x(T)$

RL update $\Delta\theta = \eta \sum_t \left\langle R \dfrac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle$

$a_t = \mathrm{up\,or\,down}$

$$\frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} = \pm e^{-\theta} \pi_\theta(a_t) = \pm(1-\pi_\theta(a_t)) = \begin{matrix} 1-\pi_\theta(\mathrm{up}) \\ -\pi_\theta(\mathrm{up}) \end{matrix}$$

+ for up, - for down

$1-\pi_\theta(\mathrm{up})$ for up

$-\pi_\theta(\mathrm{up})$ for down

$$\sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} = N_{\mathrm{up}} - N\pi_\theta(\mathrm{up})$$

number of 'up-steps'

N=number of time steps

The simplest RL example ever

return $R = x(T) = N_{\rm up} - N_{\rm down} = 2N_{\rm up} - N$

RL update $\Delta\theta = \eta \sum_t \left\langle R \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle$

$a_t = \text{up or down}$

$$\left\langle R \sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle = 2 \left\langle (N_{\rm up} - \frac{N}{2})(N_{\rm up} - \bar{N}_{\rm up}) \right\rangle$$

(general analytical expression for average update, rare)

Initially, when $\pi_\theta(\text{up}) = \frac{1}{2}$:

$$\Delta\theta = 2\eta \left\langle (N_{\rm up} - \frac{N}{2})^2 \right\rangle = 2\eta \text{Var}(N_{\rm up}) = \eta \frac{N}{2} > 0$$

(binomial distribution!)

The simplest RL example ever

In general:

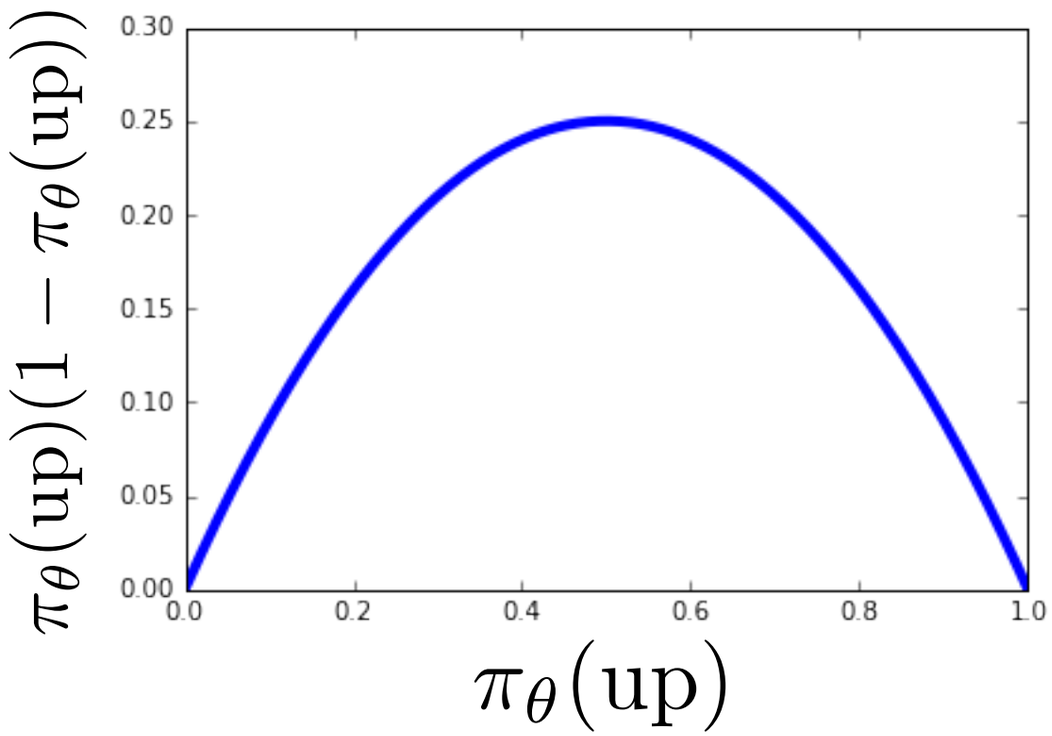
$$\left\langle R \sum_t \frac{\partial \ln \pi_\theta(a_t)}{\partial \theta} \right\rangle = 2 \left\langle (N_{\rm up} - \frac{N}{2})(N_{\rm up} - \bar{N}_{\rm up}) \right\rangle$$

$$= 2 \left\langle \left((N_{\rm up} - \bar{N}_{\rm up}) + (\bar{N}_{\rm up} - \frac{N}{2}) \right) (N_{\rm up} - \bar{N}_{\rm up}) \right\rangle$$

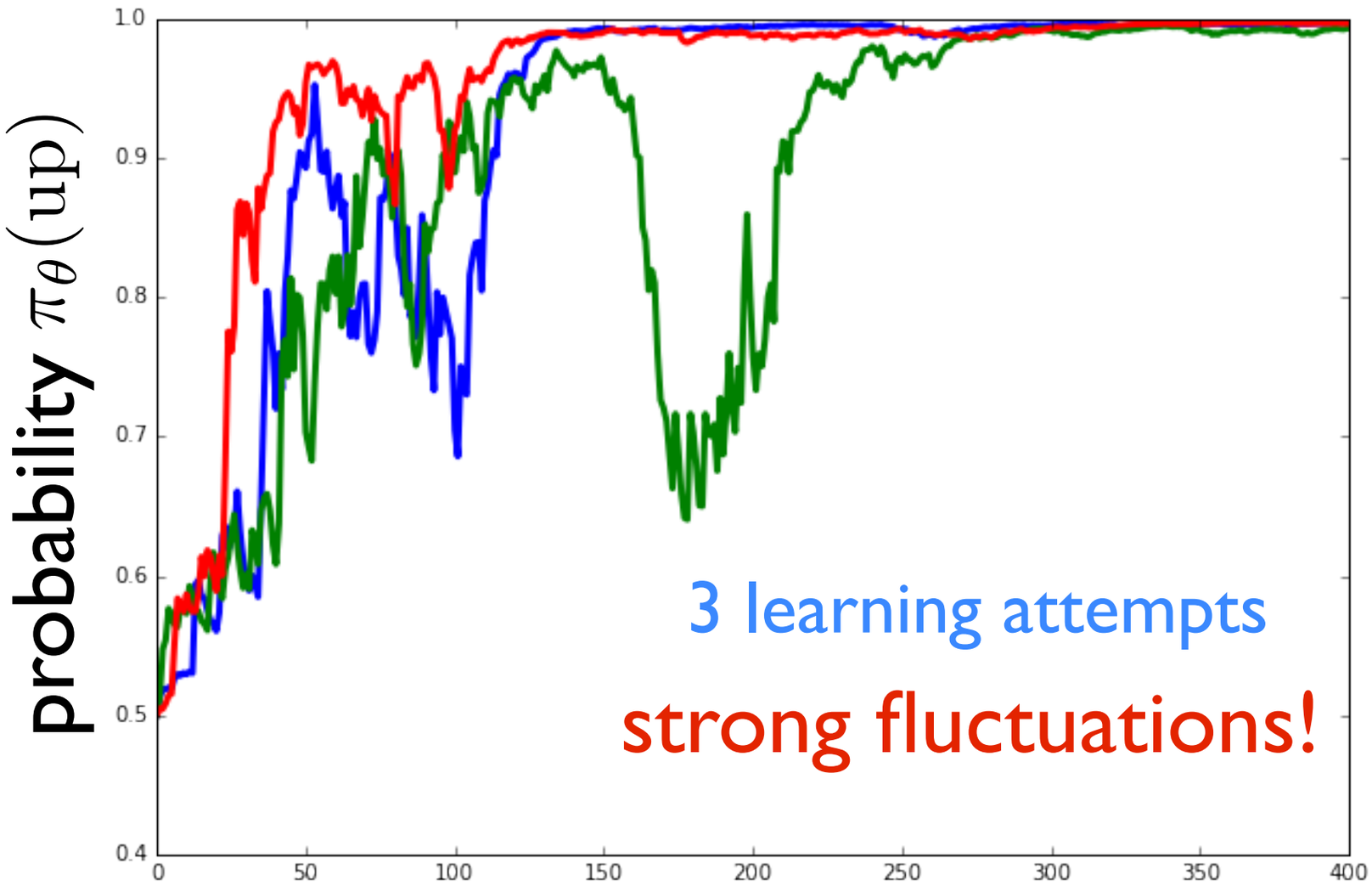
$$= 2 {\rm Var} N_{\rm up} + 2(\bar{N}_{\rm up} - \frac{N}{2}) \left\langle N_{\rm up} - \bar{N}_{\rm up} \right\rangle$$

$$= 2 {\rm Var} N_{\rm up} = 2 N \pi_\theta({\rm up})(1 - \pi_\theta({\rm up}))$$

(general analytical expression for average update, fully simplified, extremely rare)



The simplest RL example ever



trajectory (=training episode)

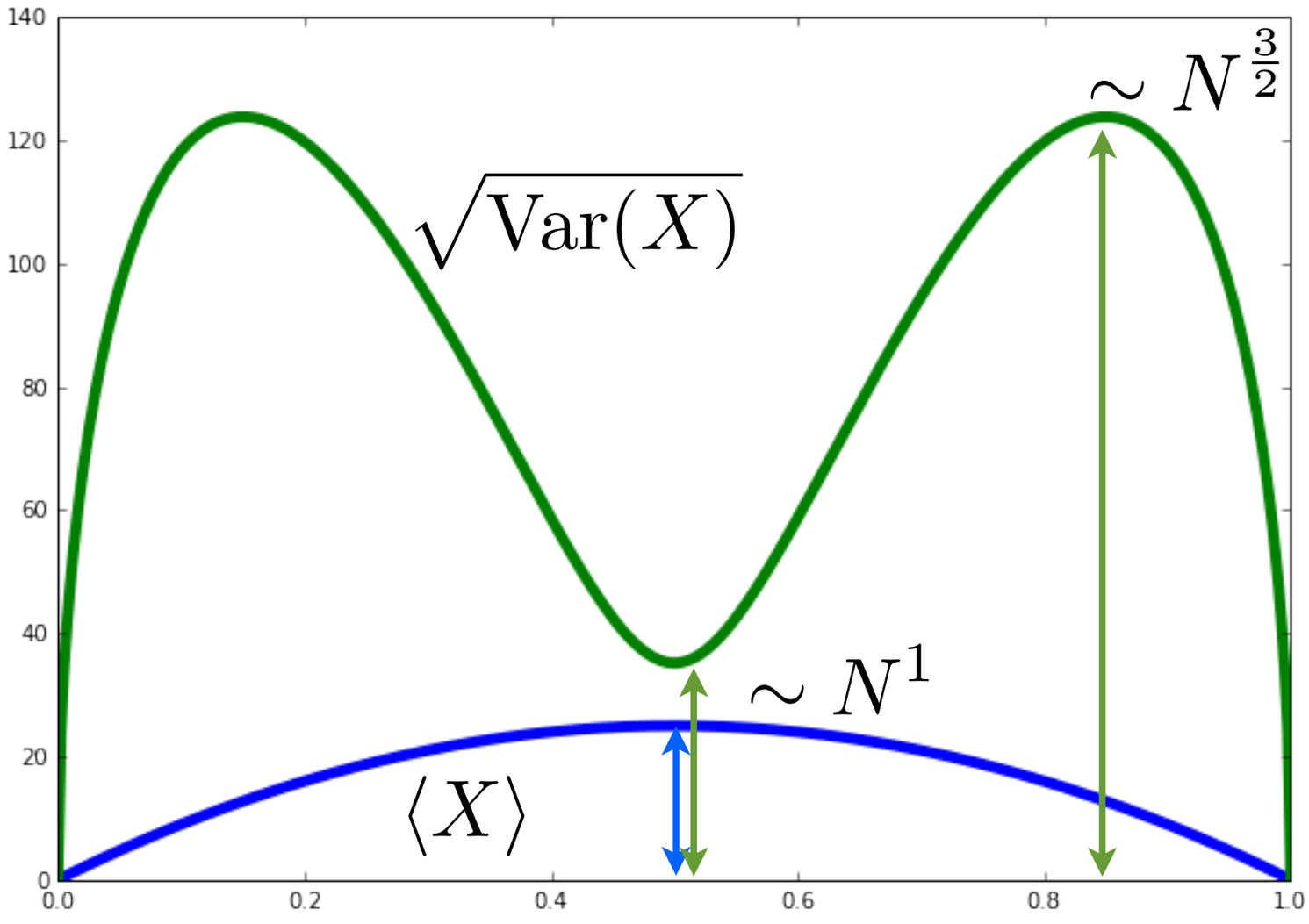
(This plot for N=100 time steps in a trajectory; eta=0.001)

Spread of the update step

$$Y = N_{\mathrm{up}} - \bar{N}_{\mathrm{up}} \qquad c = \bar{N}_{\mathrm{up}} - N/2 \qquad X = (Y + c)Y$$

X=update (except prefactor of 2)

(Note: to get Var X, we need central moments of binomial distribution up to 4th moment)



$\pi_\theta(\mathrm{up})$

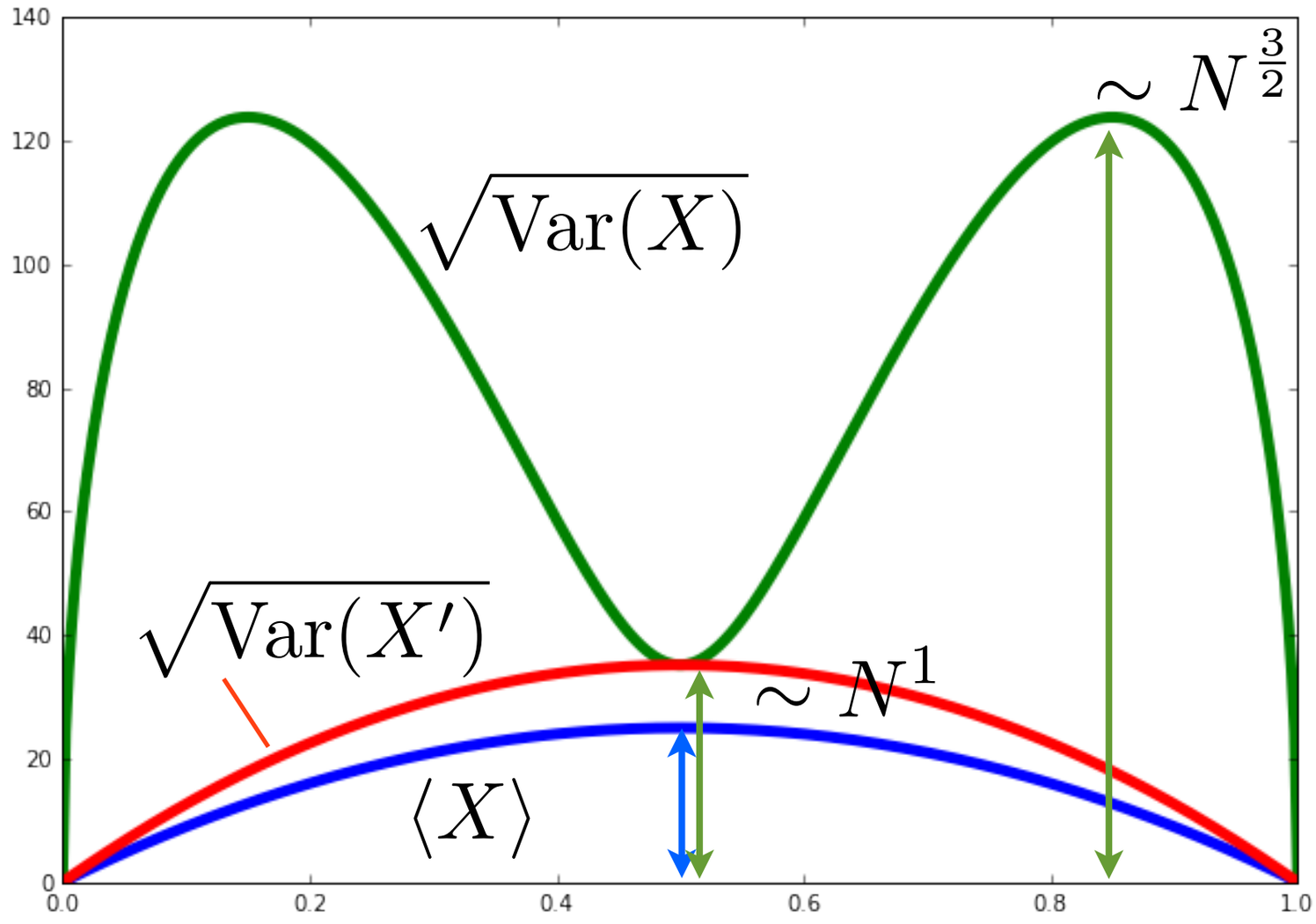
(This plot for N=100)

Optimal baseline suppresses spread!

$$Y = N_{\mathrm{up}} - \bar{N}_{\mathrm{up}} \qquad c = \bar{N}_{\mathrm{up}} - N/2 \qquad X = (Y + c)Y$$

with optimal baseline:

$$X' = (Y + c - b)Y \qquad b = \frac{\langle Y^2(Y + c)\rangle}{\langle Y^2\rangle}$$



(This plot for N=100)

Note: Many update steps reduce relative spread

M = number of update steps

$$\Delta X = \sum_{j=1}^{M} X_j$$

$$\langle \Delta X \rangle = M \langle X \rangle$$

$$\sqrt{\mathrm{Var}\Delta X} = \sqrt{M}\sqrt{\mathrm{Var}X}$$

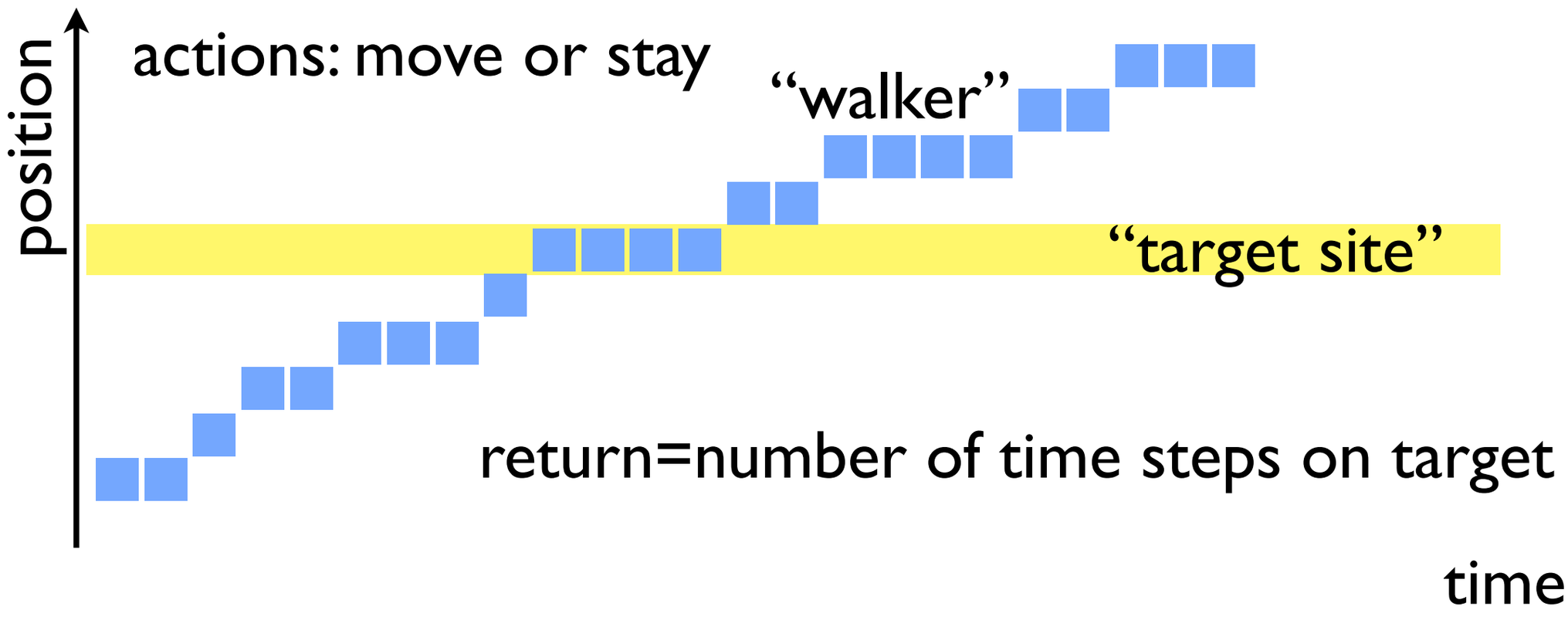
relative spread $$\frac{\sqrt{\mathrm{Var}\Delta X}}{\langle \Delta X \rangle} \sim \frac{1}{\sqrt{M}}$$

Homework

Implement the RL update including the optimal baseline and run some stochastic learning attempts. Can you observe the improvement over the no-baseline results shown here?

Note: You do not need to simulate the individual random walk trajectories, just exploit the binomial distribution.

The second-simplest RL example



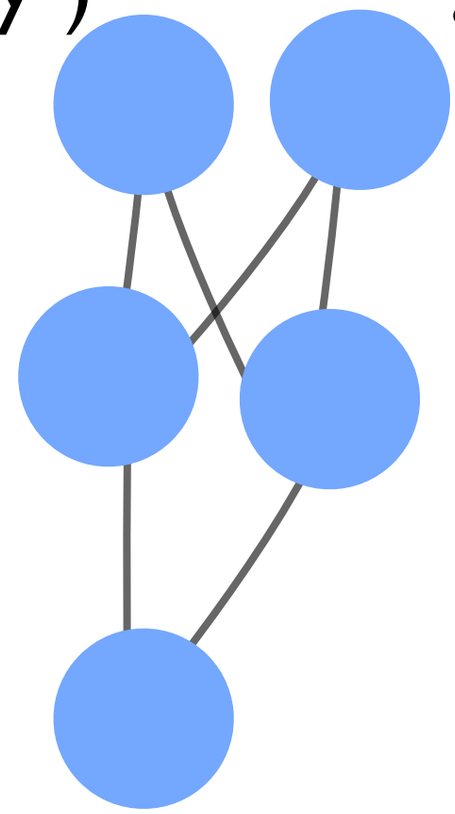
See code on website: "SimpleRL_WalkerTarget"

output = action probabilities (softmax)

policy $\pi_\theta(a|s)$

a=0 ("stay")

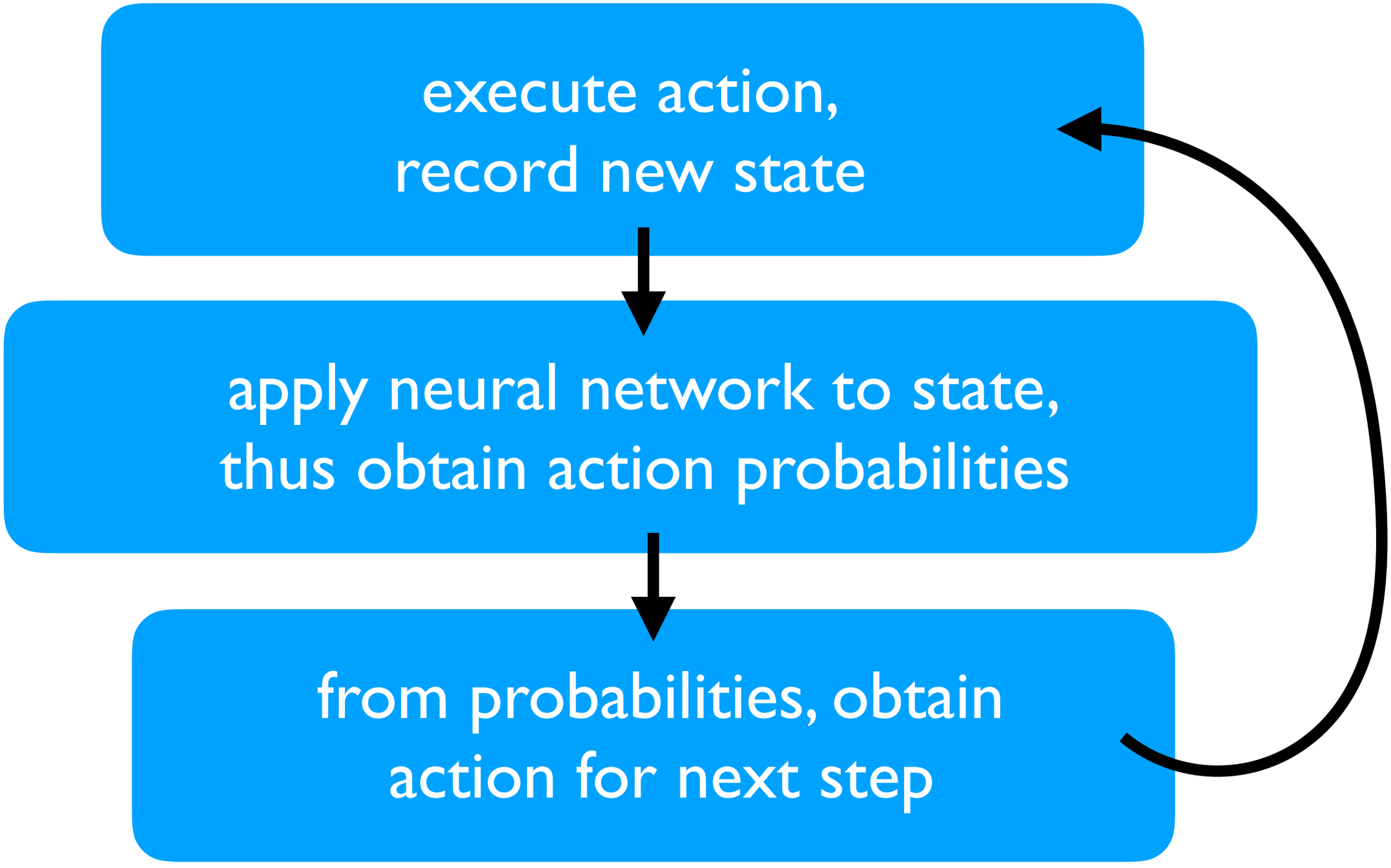
a=1 ("move")



input = s = "are we on target"? (0/1)

Policy gradient: all the steps

Obtain one "trajectory":



Policy gradient: all the steps

For each trajectory:

Do one trajectory
(in reality: a batch of trajectories)

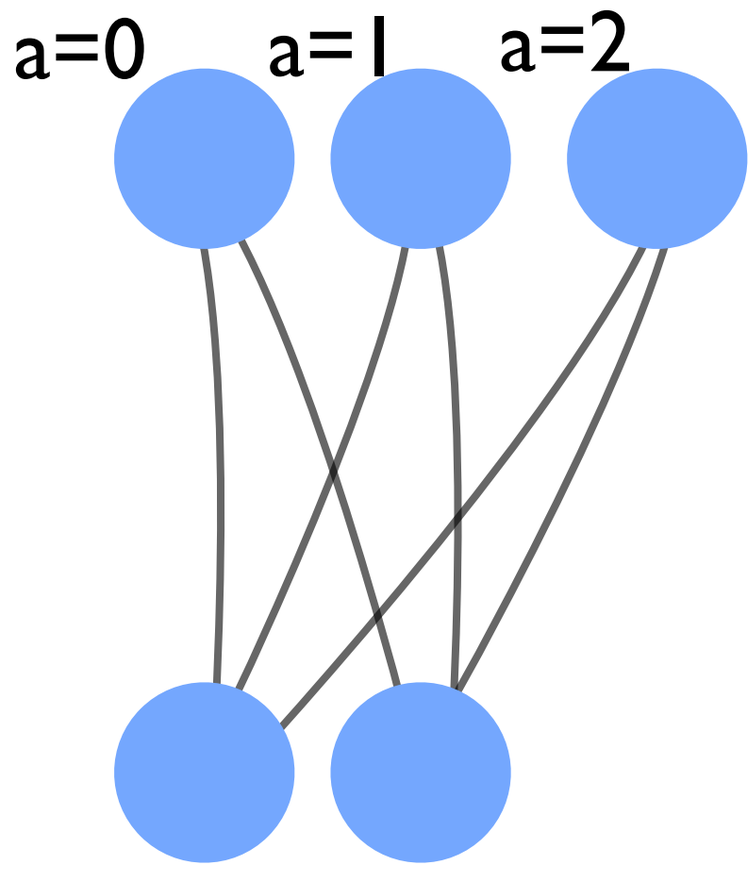
Obtain overall sum of rewards (=return)
for each trajectory

apply policy gradient training
(enhance probabilities for all
actions in a high-return trajectory)

RL in keras: categorical cross-entropy trick

output = action probabilities (softmax)

$$\pi_\theta(a|s)$$



categorical cross-entropy

$$C = -\sum_a P(a) \ln \pi_\theta(a|s)$$

distr. from net

desired distribution

Set

$$P(a) = R$$

for a=action that was taken

$$P(a) = 0$$

for all other actions a

$$\Delta\theta = -\eta \frac{\partial C}{\partial \theta}$$

implements policy gradient

RL in keras: categorical cross-entropy trick

Encountered N states (during repeated runs)

After setting categorical cross-entropy as cost function, just use the following simple line to implement policy gradient:

array N x state-size

```
net.train_on_batch(observed_inputs,desired_outputs)
```

array N x number-of-actions

Here **desired_outputs[j,a]=R** for the state numbered **j**, if action **a** was taken during a run that gave overall return **R**

AlphaGo

Among the major board games, “Go” was not yet played on a superhuman level by any program (very large state space on a 19x19 board!)

alpha-Go beat the world’s best player in 2017

AlphaGo

First: try to learn from human expert players

sampled state-action pairs (s, a), using stochastic gradient ascent to maximize the likelihood of the human move a selected in state s

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial\sigma}$$

We trained a 13-layer policy network, which we call the SL policy network, from 30 million positions from the KGS Go Server. The net-

Silver et al.,"Mastering the game of Go with deep neural networks and tree search" (Google Deepmind team), Nature, January 2016

AlphaGo

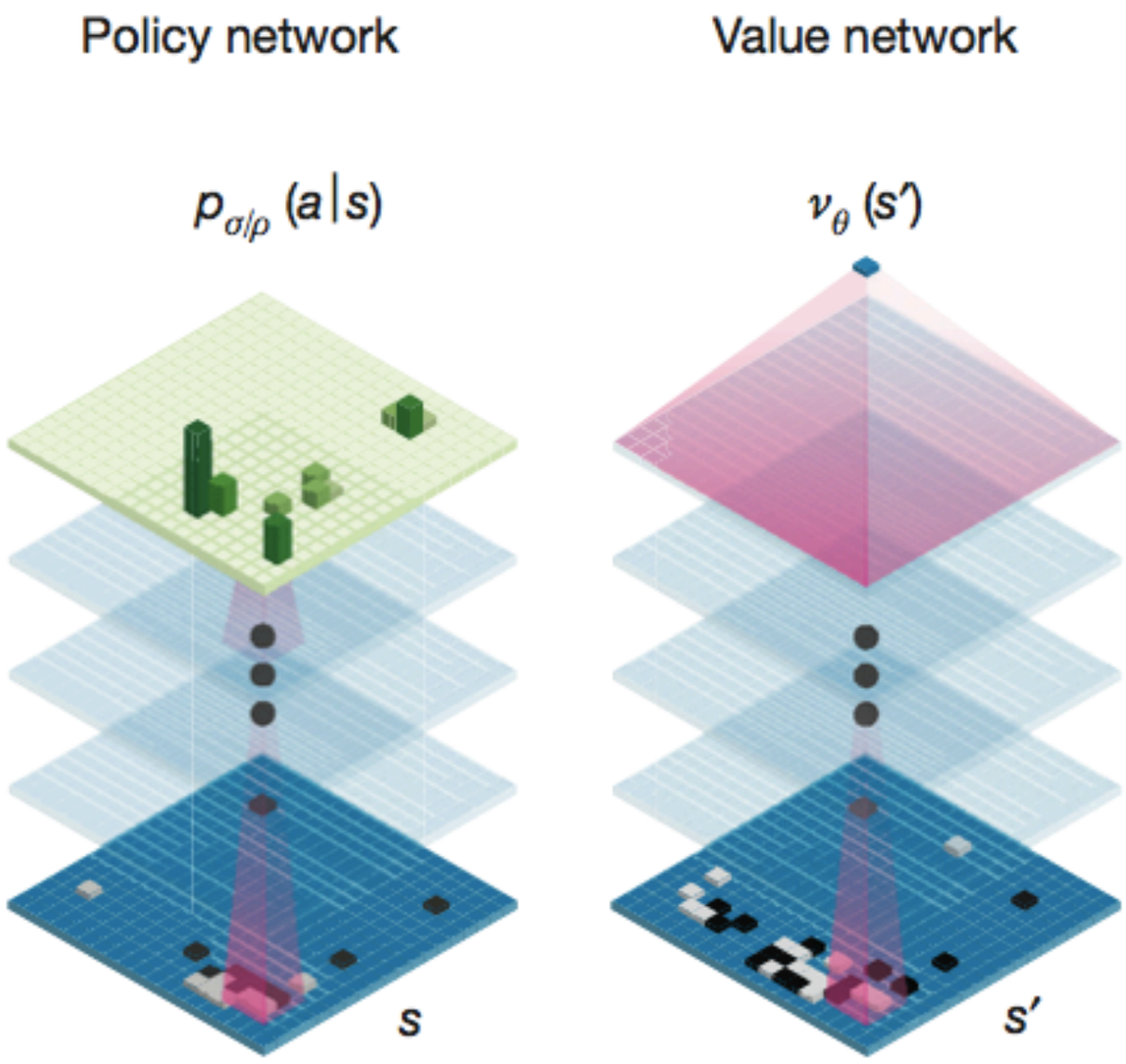
Second: use policy gradient RL on games played against previous versions of the program

to the current policy. We use a reward function $r(s)$ that is zero for all non-terminal time steps $t < T$. The outcome $z_t = \pm r(s_T)$ is the terminal reward at the end of the game from the perspective of the current player at time step t: $+1$ for winning and -1 for losing. Weights are then updated at each time step t by stochastic gradient ascent in the direction that maximizes expected outcome[25]

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$$

Silver et al.,"Mastering the game of Go with deep neural networks and tree search" (Google Deepmind team), Nature, January 2016

AlphaGo

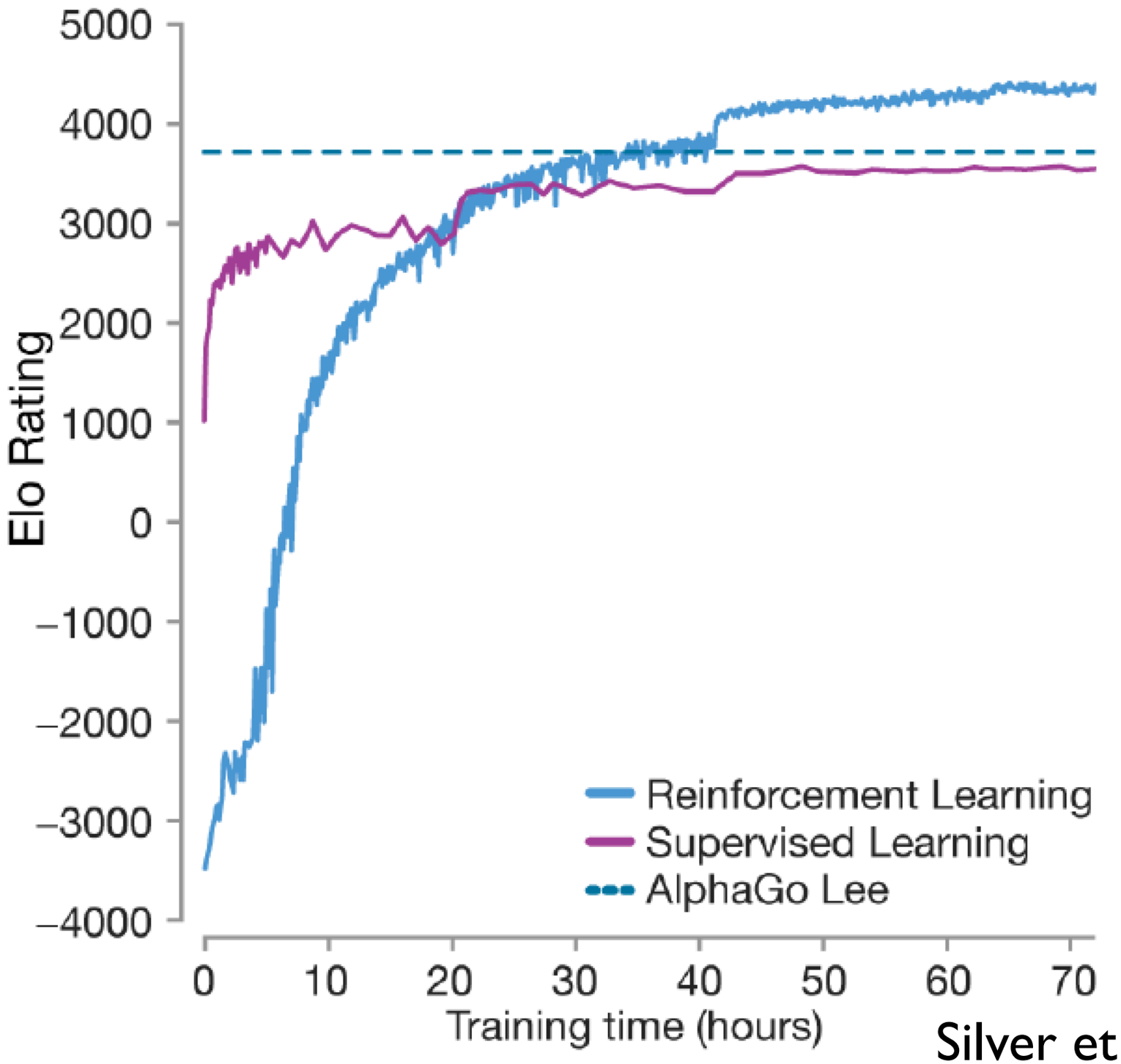


*Note: beyond policy-gradient type methods, this also includes another algorithm, called Monte Carlo Tree Search

Silver et al.,"Mastering the game of Go with deep neural networks and tree search" (Google Deepmind team), Nature, January 2016

AlphaGoZero

No training on human expert knowledge
– eventually becomes even better!



Silver et al, Nature 2017

AlphaGoZero

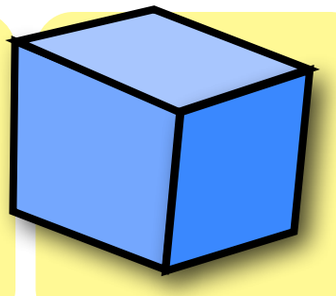
Ke Jie stated that "After humanity spent thousands of years improving our tactics, computers tell us that humans are completely wrong... I would go as far as to say not a single human has touched the edge of the truth of Go."

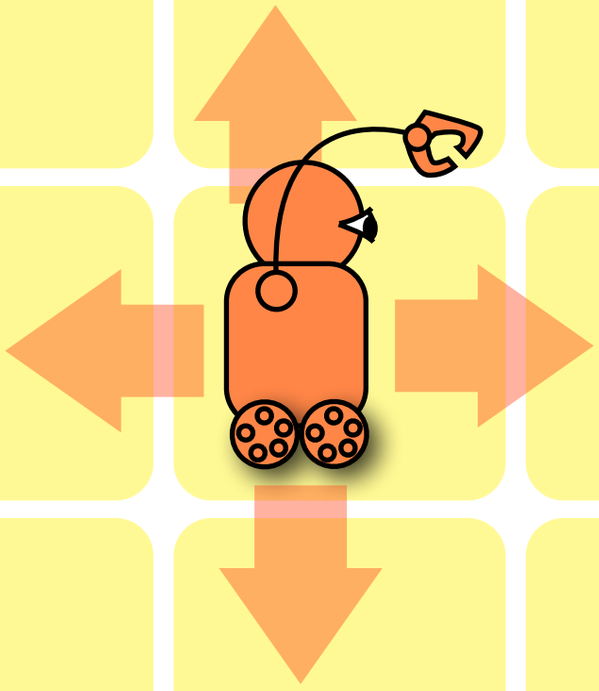
Q-learning

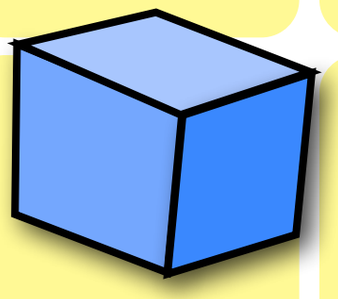
Q-learning

An alternative to the policy gradient approach

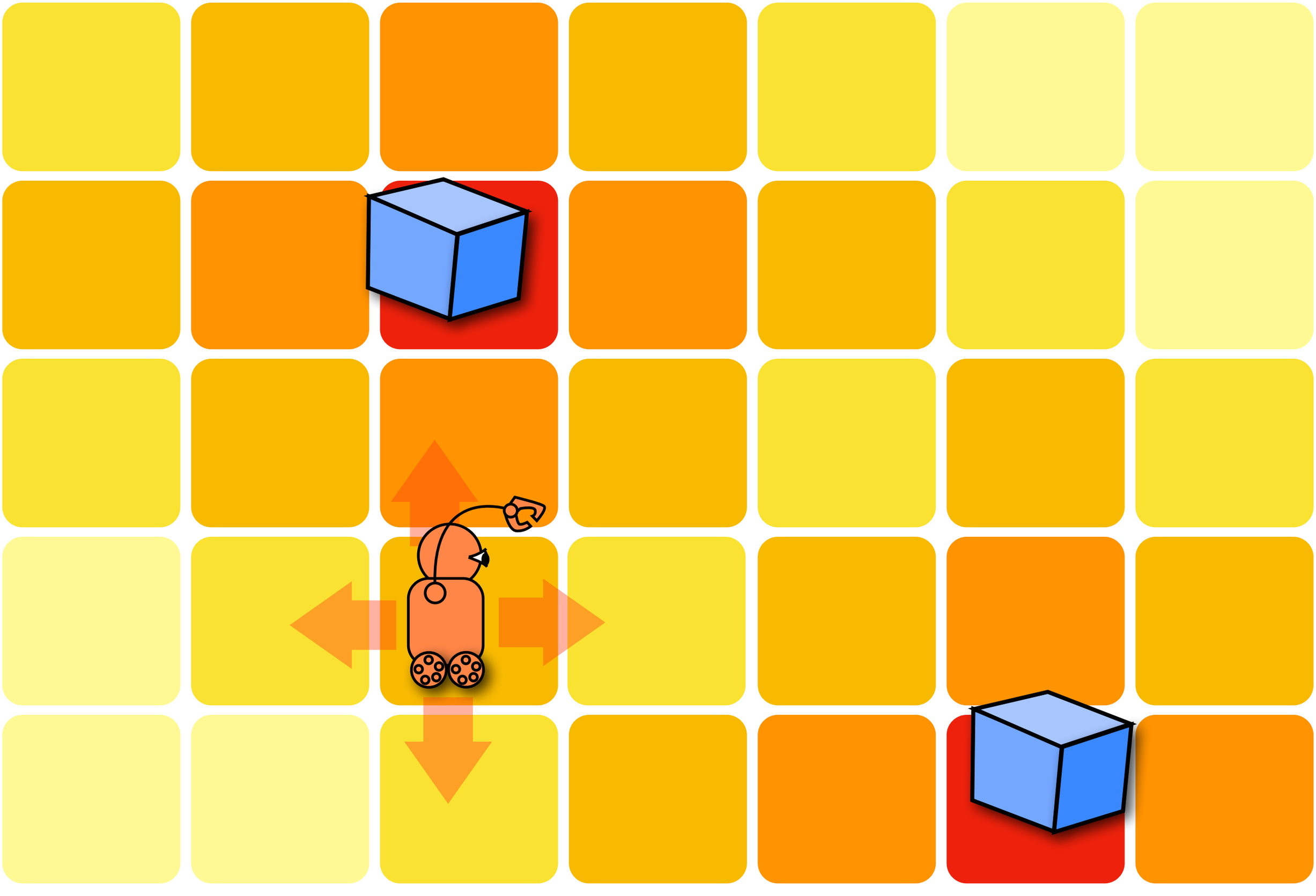
Introduce a quality function Q that predicts the future reward for a given state s and a given action a. **Deterministic policy**: just select the action with the largest Q!



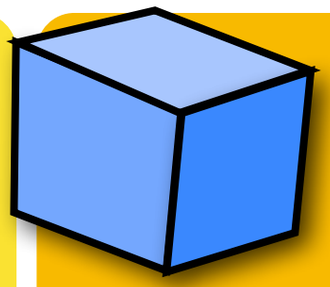


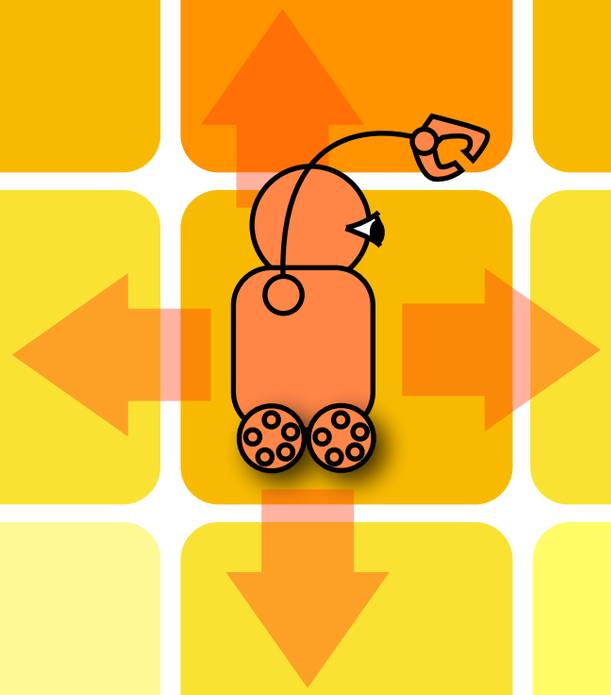


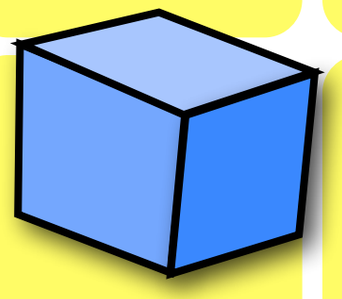
"value" of a state as color



"quality" of the action "going up" as color







Q-learning

Introduce a quality function Q that predicts the future reward for a given state s and a given action a. **Deterministic policy**: just select the action with the largest Q!

$$Q(s_t, a_t) = E[R_t | s_t, a_t]$$

(assuming future steps to follow the policy!)

"Discounted" future reward:

$$R_t = \sum_{t'=t}^{T} r_{t'} \gamma^{t'-t}$$

Reward at time step t: r_t

depends on state and action at time t

Discount factor: $0 < \gamma \leq 1$

learning somewhat easier for smaller factor (short memory times)

Note: The 'value' of a state is $V(s) = \max_a Q(s, a)$

How do we obtain Q?

Q-learning: Update rule

Bellmann equation: (from optimal control theory)

$$Q(s_t, a_t) = E[r_t + \gamma \max_a Q(s_{t+1}, a) | s_t, a_t]$$

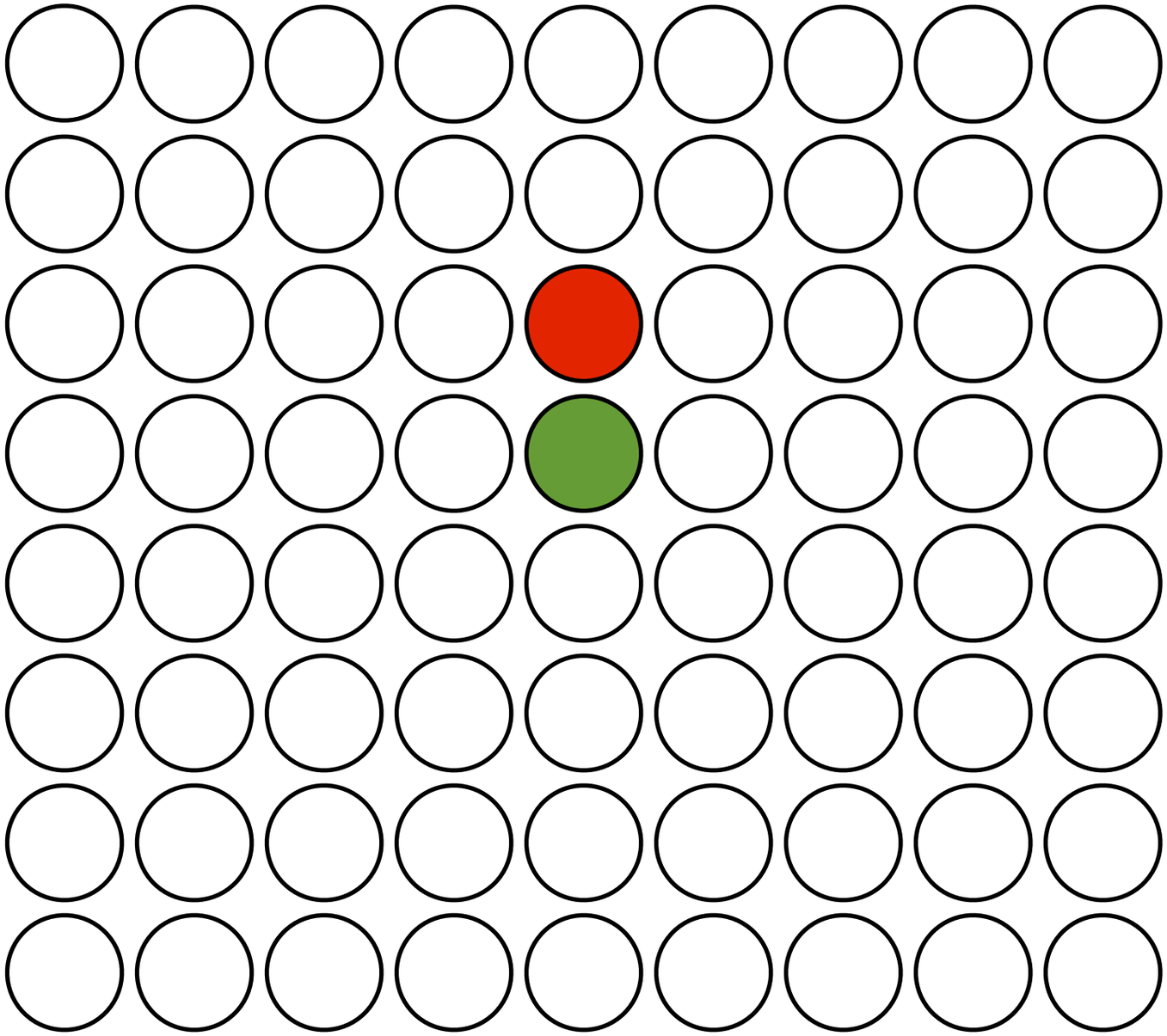
In practice, we do not know the Q function yet, so we cannot directly use the Bellmann equation. However, the following update rule has the correct Q function as a fixed point:

$$Q^{\text{new}}(s_t, a_t) = Q^{\text{old}}(s_t, a_t) + \alpha(r_t + \gamma \max_a Q^{\text{old}}(s_{t+1}, a) - Q^{\text{old}}(s_t, a_t))$$

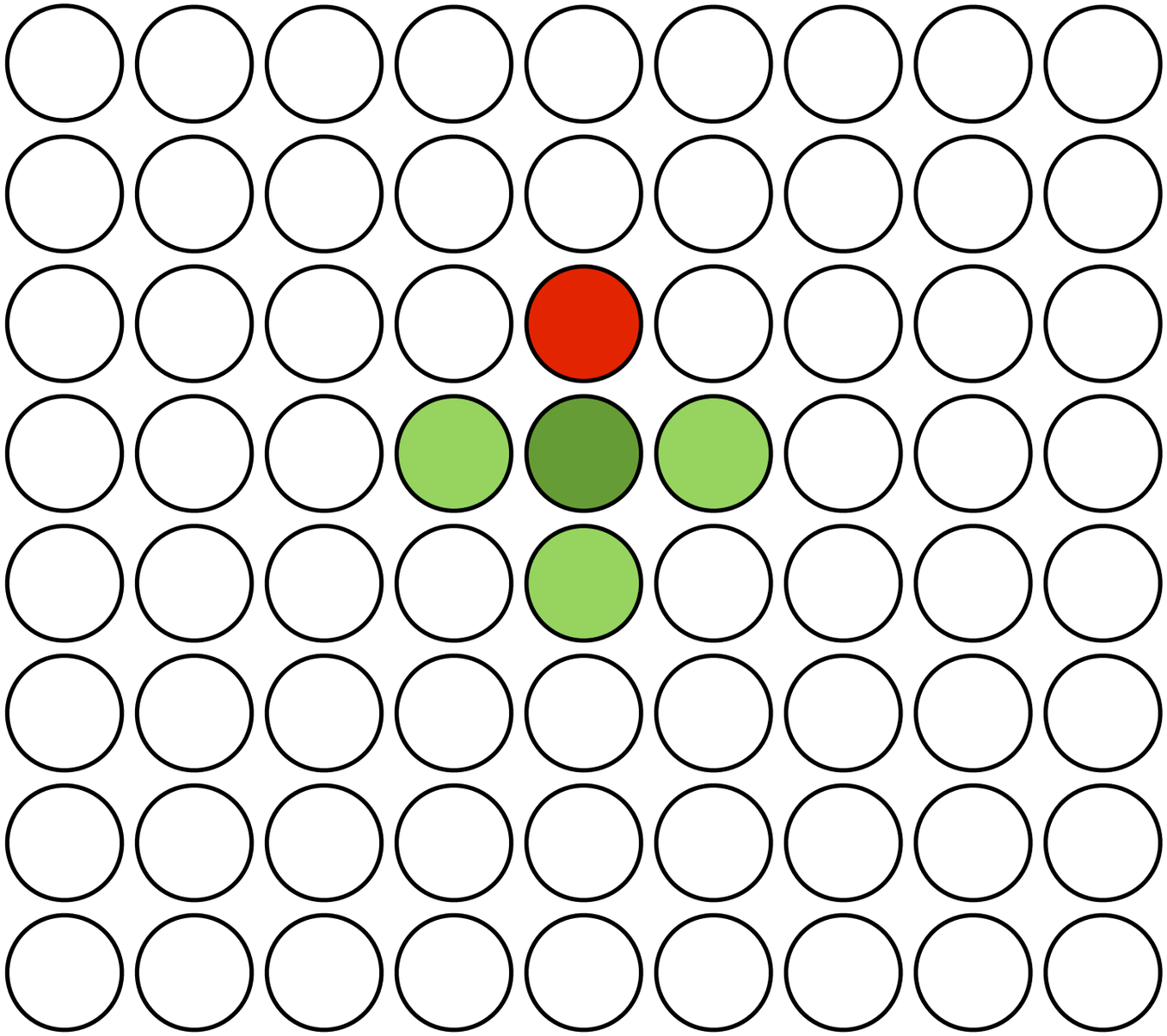
small (<1) update factor

will be zero, once we have converged to the correct Q

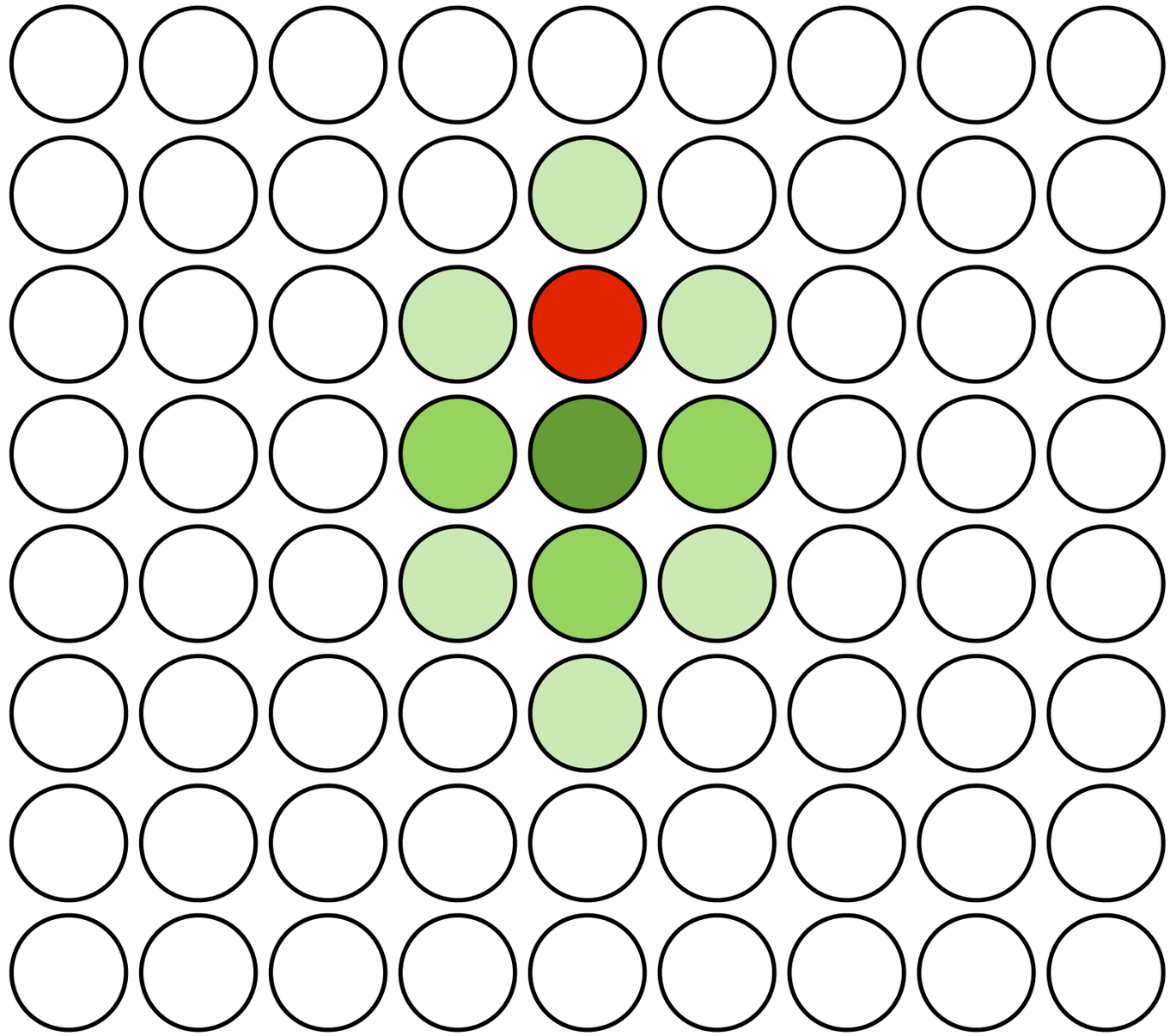
If we use a neural network to calculate Q, it will be trained to yield the "new" value in each step.



Q(a=up,s)



Q(a=up,s)



Q(a=up,s)

Q-learning: Exploration

Initially, Q is arbitrary. It will be bad to follow this Q all the time. Therefore, introduce probability ϵ of random action ("exploration")!

Follow Q: "**exploitation**"

Do something random (new): "**exploration**"

"ϵ-greedy"

Reduce this randomness later!

Example: Learning to play Atari Video Games

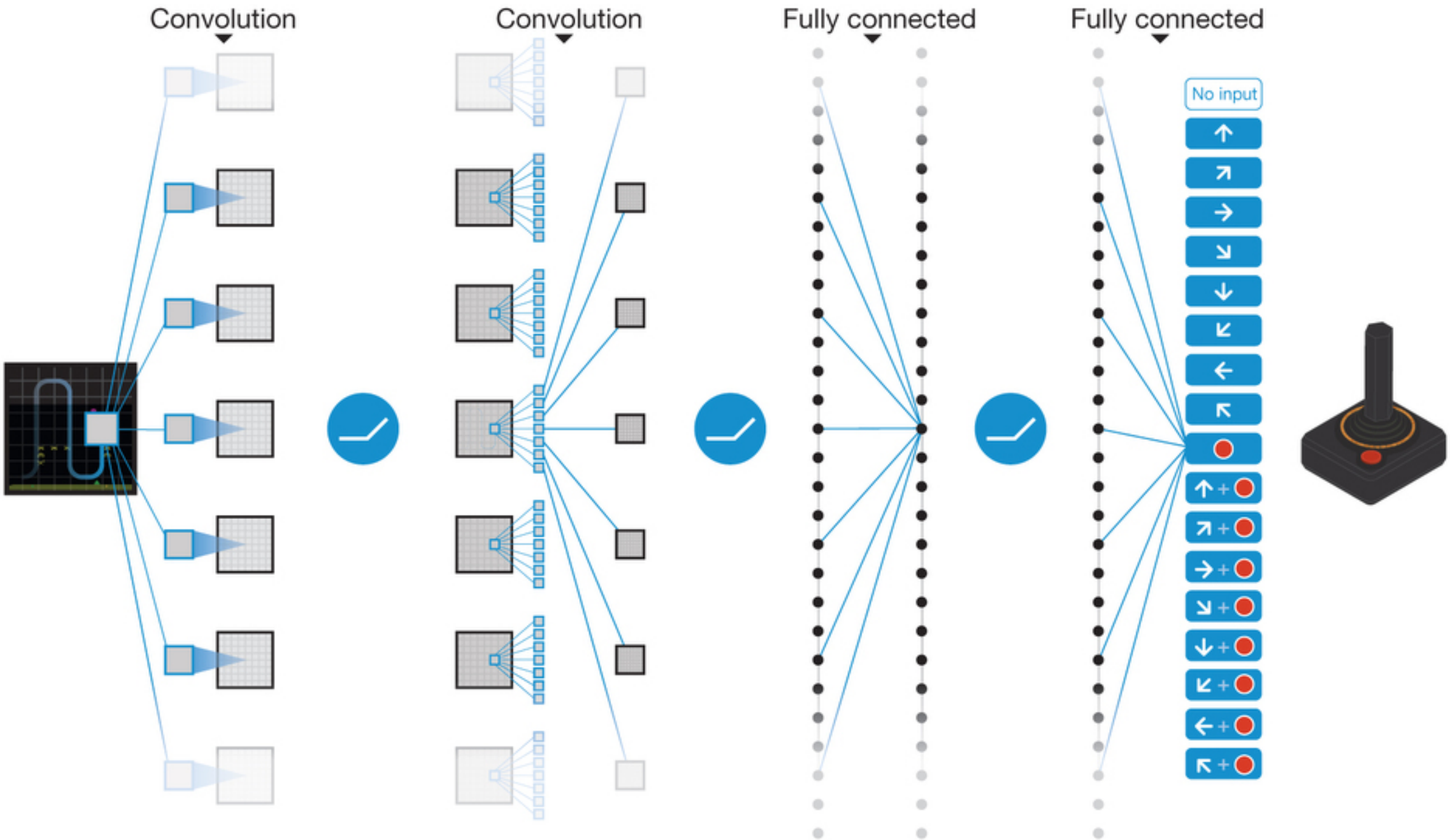
"Human-level control through deep reinforcement learning", Mnih et al., Nature, February 2015




last four 84x84 pixel images as input [=state]
motion as output [=action]

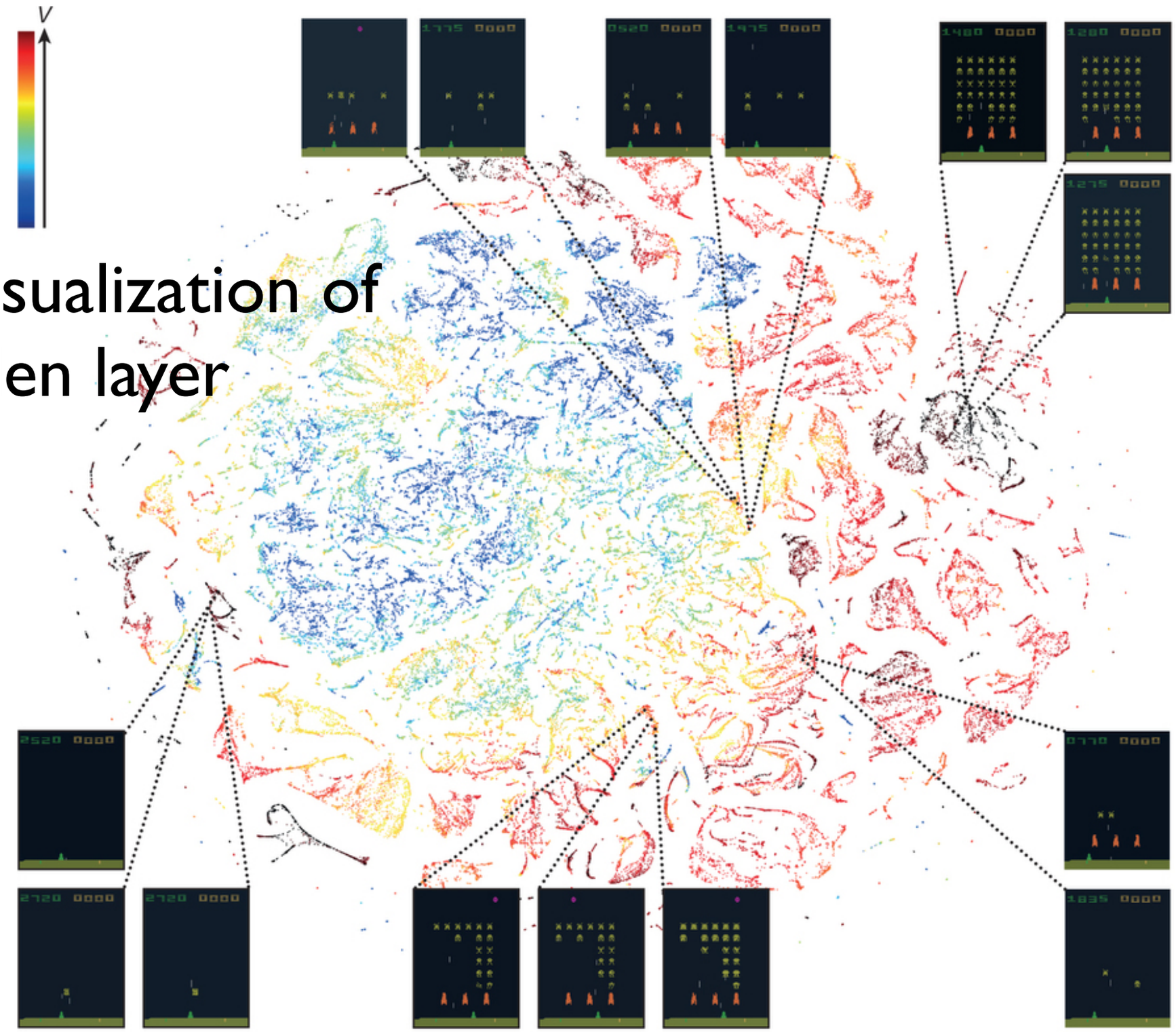
Example: Learning to play Atari Video Games

"Human-level control through deep reinforcement learning", Mnih et al., Nature, February 2015



Example: Learning to play Atari Video Games

"Human-level control through deep reinforcement learning", Mnih et al., Nature, February 2015



t-SNE visualization of last hidden layer

Advanced Exercise

Apply RL to solve the challenge of finding, as fast as possible, a "treasure" in:

- a fixed given labyrinth
- an arbitrary labyrinth (in each run, the player finds itself in another labyrinth)

Use the labyrinth generator on Wikipedia "Maze Generation Algorithm"

Wikipedia: "Maze Generation Algorithm / Python Code Example"